Free Range Rovers

White Paper

04.08.2021

Emily Brantner, Blair Jones, Jai Raju
UC Berkeley MIDS

Current robo-mowers require initial work to get them started, whether it's walking the machine around your yard or installing underground wires for edge detection. We propose an alternative: a robot that learns the edges without any more input from you than a few taps on your phone screen. We're on a quest to take the "work" out of yard work.

Overview

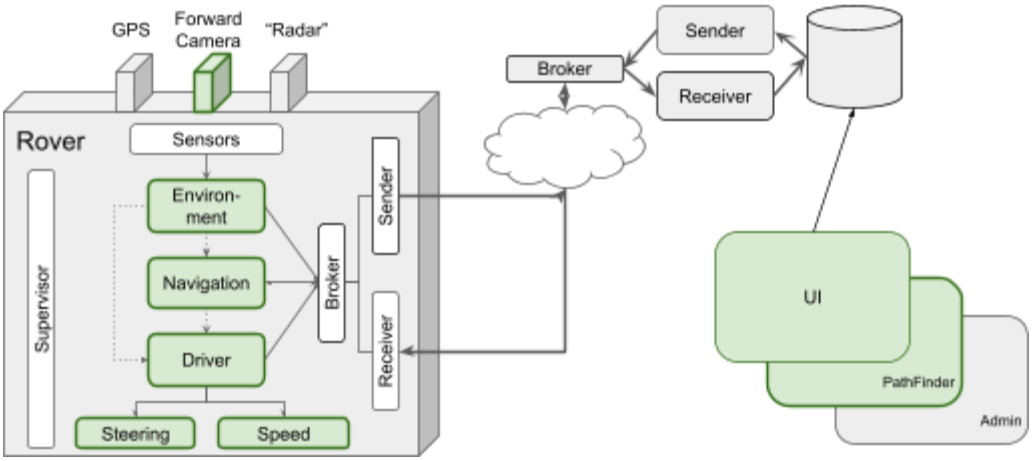
We set out to build a mower that takes the work out of setup, and creates an almost completely hands-off and seamless experience for the user. All the user needs to do is supply their address, mark the areas of the yard that they want to be mowed, and choose the placement of their mower. After that, our machine learning algorithms do the rest of the work!

Project Goals:

- From an overhead picture, detect the navigable spaces for the Rover to mow.
- From those predictions, allow user input to alter navigable space as necessary, and mark the "home" location of their Rover.
- Build version zero of the Rover (no mowing yet)
- Train a computer vision model to detect objects that should be avoided
- Combine the computer vision model with the overhead prediction model to navigate a path while avoiding both objects and non-navigable spaces.

How does it work?

Our project is split into two main parts- **Detecting Navigable Spaces** and **Autonomous Navigation**. Detecting navigable spaces breaks down into smaller chunks: model training, image pre-processing, running the model, and user input. Autonomous navigation breaks down into building the rover, training the model, and running the algorithm.



Detect navigable spaces

We start with an address, provided by a user or using location services on a device. We use an aerial image of the terrain around the address from available data sources (ex. https://gis.dupageco.org/parcelviewer/). We then use the overhead image to detect and mark the different types of ground or ground cover in the image, such as:

- Navigable: grass, pathway
- Non-navigable: gravel, trees, house, etc.

This is shown to the user so that they may manually override to add or remove navigable areas. The user may also place the home location, or docking station, of their rover onto the map.

The marked-up map is then exported as a json file to be used in the autonomous navigation system on the rover itself.

Our Approach

Labeled Training Data
- Semantic segmentation for navigable (grass, driveway) vs non-navigable (gravel, shrubs, trees, fence, etc.)

Created Augmentations
- Rotation
- Mirror
- Crop
- Pixelation
- Shift

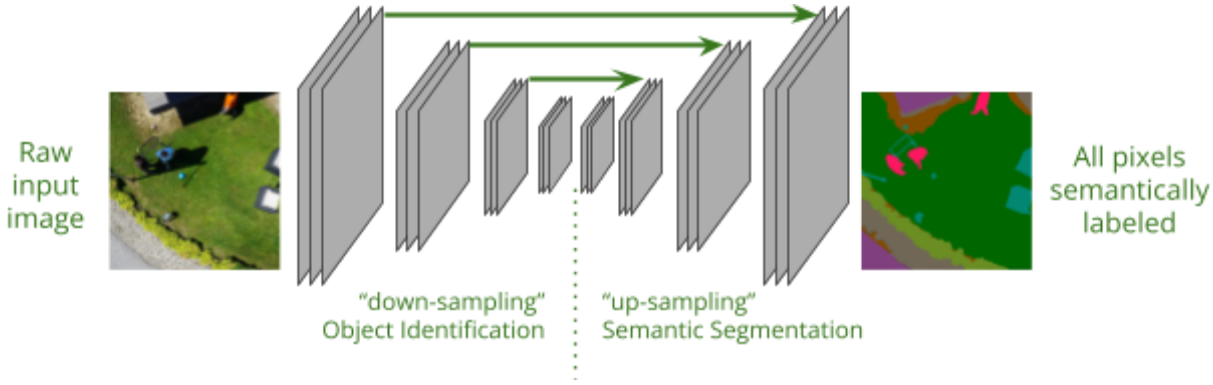
Training Runs
- U-Net selection
- Random Image Augmentation Types
- Hyperparameter selection

Detect Navigable Spaces
- User-provided address
- Infer target spaces from aerial image of location
- User has option to add labels / detail to inferred map

Publish to devices
- Pipeline to pull map of navigable space to edge device

Semantic Detection

Our model for detecting navigable spaces uses the U-Net architecture for semantic segmentation. Specifically, it's split into two main parts: down-sampling for object identification and up-sampling for semantic segmentation. The model uses the MobileNetV2 architecture for the downsampling portion and was initialized with ImageNet weights. The upsampling portion was inspired by the Pix2Pix implementation and was initialized with random weights. In our neural net, ReLU activation layers were used. After comparing performance between freezing or training the down-sample layers, we chose to freeze them, as the resulting accuracy was adequate and training was faster. In the U-Net architecture approach, "skip connections" are established from the downsampling outputs to the upsampling activations. Then, we were able to package the inference model separately so that we can run overhead views of actual properties through it to determine their navigable space.



Training the Model

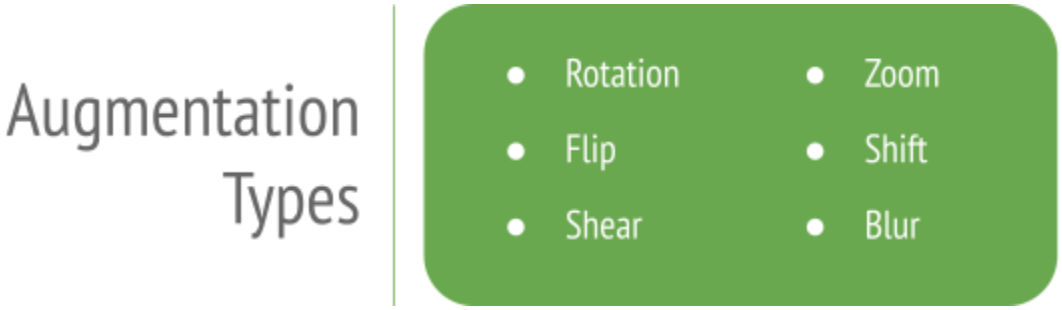
To train the model, we first needed a significant number of labeled low-altitude overhead images with decent enough resolution to distinguish between navigable areas, such as grass or driveway, and to non-navigable areas, such as shrubs, gravel, or flowerbeds.

We initially worked on manually extracting overhead images from freely available map sources, primarily county-level property websites. We labeled the extracted images, aerial photos, using 5 different classes for house, car, grass, shrub, patio. We assigned the classes to 10-12 objects per image using boxes and drawn edges. For 20 images, that was a total of 200 labels, which took a couple of hours to complete, just for the labeling. Additional time was lost due to some issues with the labeling tool, and some format conversions that were required to switch between tools and platforms. It took about 10 hours manual labor to prepare the 20 images. Additional work was performed to augment the images and labels with transformations to generate additional training samples. A variety of custom-coded and commercial tools (ex. Roboflow) were used to try different techniques. We ended up with roughly 200 training samples after augmentations were applied.

A first pass of neural network training showed that this was not enough data and that the labeling quality needed to be significantly improved (i.e. tighter bounds around the target classes).

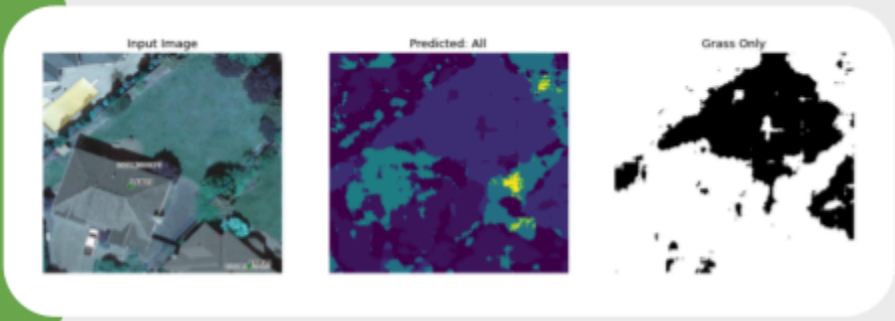
Fortunately, we located a university-developed open dataset (https://dronedataset.icg.tugraz.at) of 400 pre-augmentation residential, RGB-labeled, high res, low-altitude, drone images which were a perfect fit for our use case - determining what the ground cover is from an overhead view. A first training pass indicated that with augmentations, this dataset would fit our needs.

With the training dataset in hand, we chose to use Keras Preprocessing Layers for image augmentation. This gives us the ability to create an unlimited number of samples, and to better train our model. We visually verified a subset of the augmented images to find and address any inconsistencies in our samples (for example, blurring images too much significantly degraded training accuracy). We also cropped and resized the images from 6000 x 4000 to 224 x 224 to speed up the neural network training.



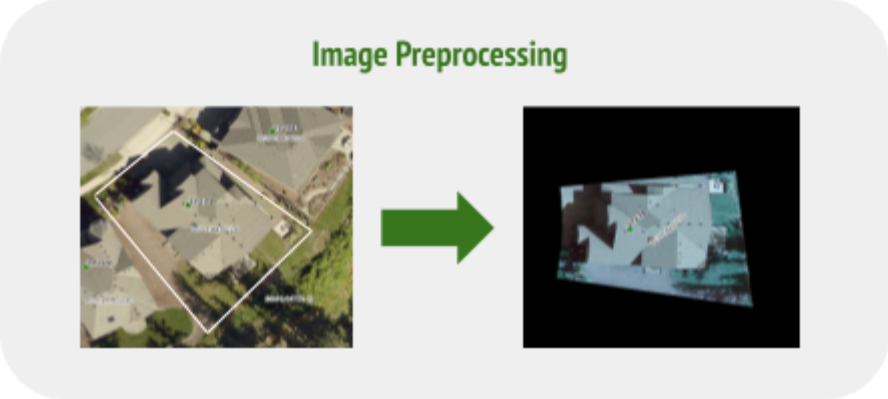
With augmented samples, we were able to begin training our model. The overall accuracy of our model after 50 epochs:

Class	Accuracy
Grass	0.90
Roof	0.91
Paved	0.89
Water	0.84



Overhead Images, Preprocessing and Running the Model

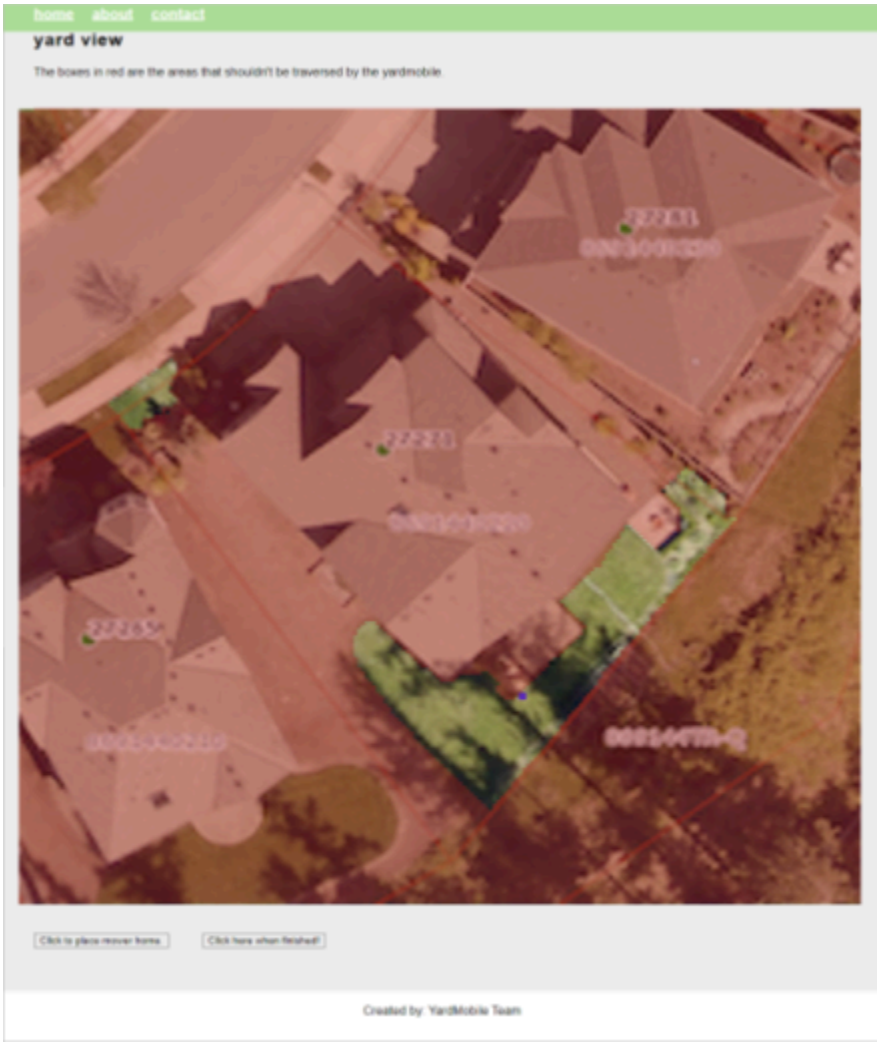
After the model was trained, we needed overhead images to test our model against the real world, and to use for our rover to detect navigable spaces. We were able to source these from county databases that showed both overhead images and property lines. We wrote python code to create a mask that covers all but the property that we're interested in. Our algorithm assumes the property of interest is in the central outlined box in the image. It then finds the property border for that area (normally indicated by red lines - but shown in white in the image below). It then masks the area outside of the property boundary, which can be used to simplify the map shown to the user. Images are also reshaped to 224x224 for input to the inference model.



Once the images have been pre-processed, then we're ready to run them through our inference model. The model outputs a 224x224 json array with different numbers labeling each type of ground cover. For our testing purposes navigable spaces were denoted with a 3 and non-navigable space with a 255.

User Input

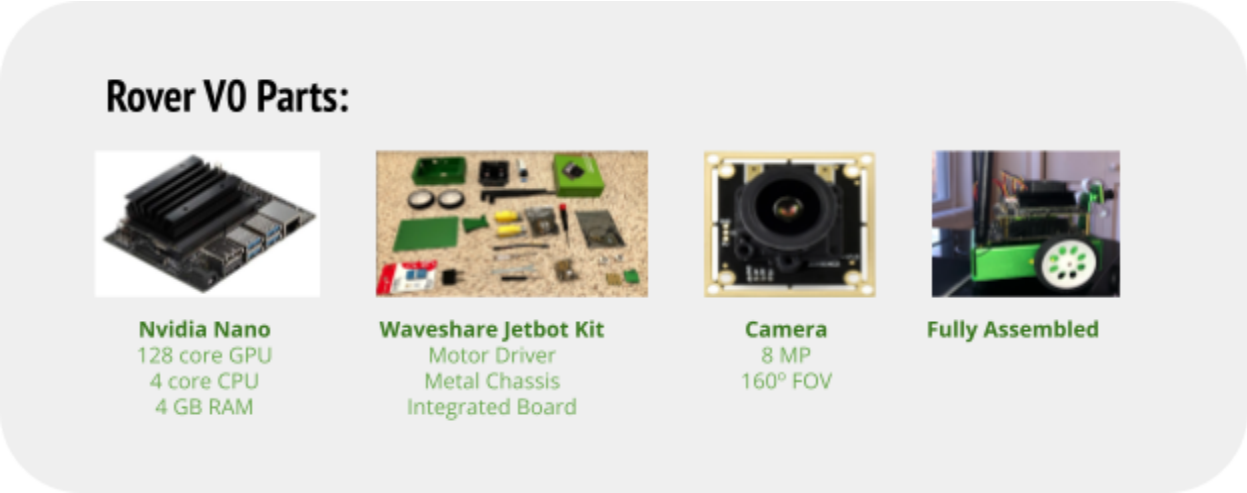
After getting the model output, the json file is ingested into the user interface, with a transparent red mask over the non-navigable areas. At this point, the user is able to click on the red boxes to make them navigable again, or click on the clear boxes to make them navigable. Then, they can mark the location of their Rover's "home" or charging station. When the user is satisfied with the navigable and non-navigable areas, they may click the "Click here when finished!" button to output a new json file, updated with their inputs. This file is then fed to the autonomous navigation system.



Autonomous Navigation

Building the Rover

For this proof of concept (POC) rover, we are primarily concerned with autonomous navigation. The POC is built from the Waveshare Jetbot AI Kit.



NVIDIA's Jetson Nano is the **brain** making all the decisions for the rover. The reasons for choosing this are:

1. The need to run a computer vision inference engine in real time
2. Due to the need of mobility, a processor that can be powered by DC
3. A need for an SoC to process the data pipeline and diving the robot

The Nano, with 128 core GPU is sufficiently supporting the cv inference engine and can be run on DC batteries. The 4 core CPU and 4GB RAM is ample capacity to run the rest of the data pipeline and drive the robot. It also came at an affordable price of $99.

Waveshare Jetbot Kit: Our goal was to focus on modeling and making an end-to-end prototype work. There were several vendors providing the individual parts to build the robot. However, we chose Waveshare's kit because it contained everything we needed for the prototype which included:

1. Chassis
2. Front facing camera
3. Shared power supply
4. Differential Motors for a 2WD
5. Motor driver

The lack of SERVO motor became an issue in the prototype which will be discussed in the What's next section.

Camera: This is the only sensor on the bot. It is a front facing wide angle camera with a 160 degrees of FOV. It captures 8 MPs images.

Building the Model

There are two schools of thought about autonomous navigation. There is the end-to-end learning approach, which is based on learning from what humans do when they drive, and the multi-sensor learning approach, similar to what Tesla does, which requires labeled data for feature extraction. The problem we are solving here is one of the few that autonomous cars solve for, like: keeping the car between the lanes, avoiding obstacles, following traffic signs etc.

In this project we are solving for *Obstacle Avoidance* or *collision avoidance* which is the most significant problem for us to solve. We solved the problem using deep learning and a

single, very versatile, sensor: the camera. We chose a logic based on two main pillars - robotics and computer vision. The outline of the code is:

1. Train a model to identify obstacles
2. Use the model to look for obstacles while driving in the path.
3. If an obstacle is found, steer around it.

Setting up an environment:

We needed a space to collect the training data and test the models. We chose a space in one of our homes where there were not many obstacles. We removed all the objects that could be removed and we simulated a realistic backyard using miniature backyard figurines. We also placed a large green plastic sheet to mimic the grass.

Data

Data collection

To collect the images we made the bot navigate to different points in the testing environment, took pictures and grouped them into Free and Blocked groups. We collected a total of 97 images for the blocked group and 103 images for the free group, some of which can be seen on the next page.

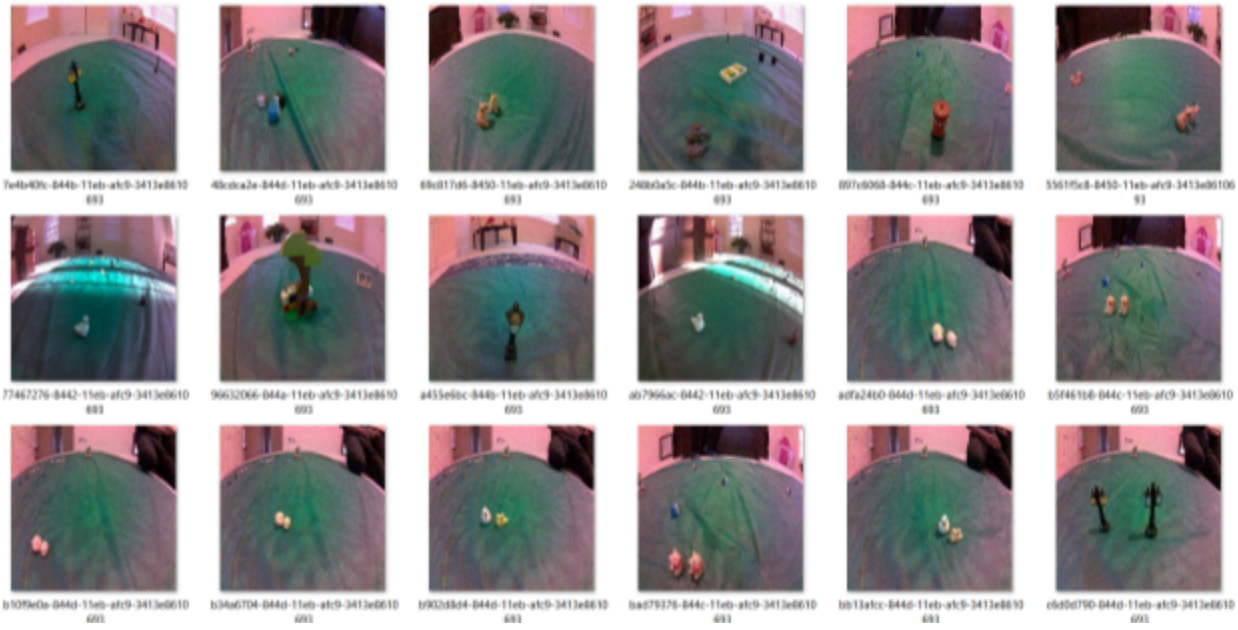
Data Preprocessing

We applied only three preprocessing steps:

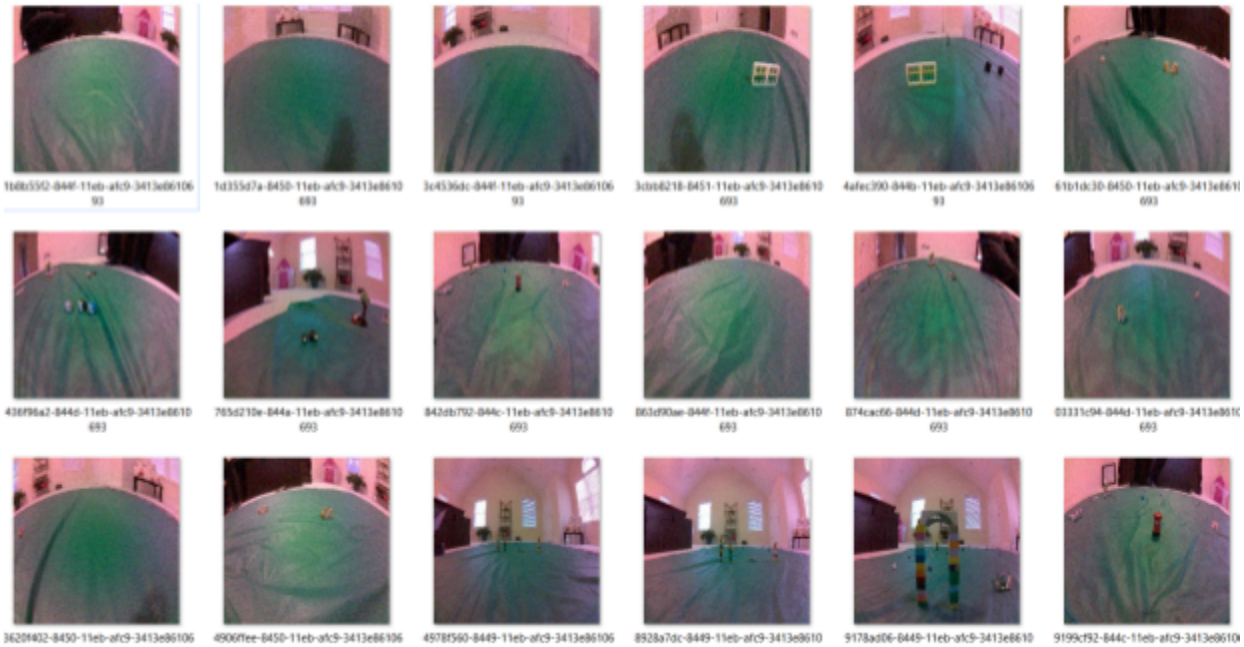
1. Color Jitter - makes a slight change in the color values of the image to simulate and generate different lighting conditions
2. Resize - the input images to 224 by 224. We didn't need to do this because our images were taken at 224 by 224. We retained it in the pipeline to give us the flexibility to include other pictures.
3. Normalize - we used the imagenet mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.225] values to normalize the images

Two Classes

Blocked Dataset

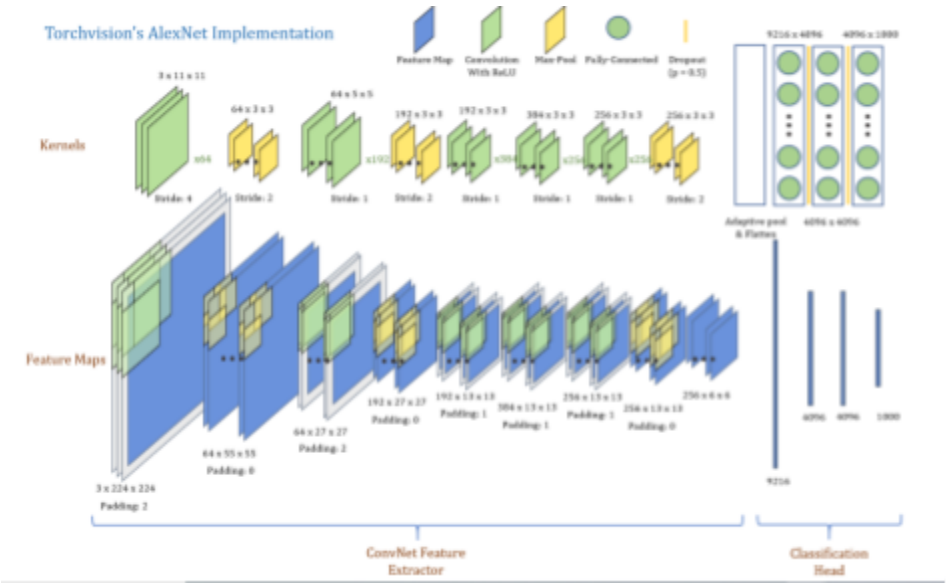


Free Dataset



Network Architecture

We are using transfer learning for feature extraction as the research on that is state-of-the-art. We used Alexnet which is trained on Imagenet and has more than a million images for image classification.



Neural Network Architecture

As we are using Pytorch for this project, we used Torchvision's implementation of Alexnet.

1. AlexNet architecture consists of 5 convolutional layers, 3 max-pooling layers, 2 normalization layers, 2 fully connected layers, and 1 softmax layer.
2. Each convolutional layer consists of convolutional filters and a nonlinear activation function ReLU.
3. The pooling layers are used to perform max pooling.

   The actual `model.eval()` output for the model is below:

```
model.eval()
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Modification

For our project, we **removed the 6th layer in the classifier and replaced it with a 2 class classifier**.

The modified network architecture is as below.

```
model.eval()
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=2, bias=True)
  )
)
```

We trained the model on the server with a NVIDIA GTX 1080, Jetson Xavier and Jetson Nano.

GTX 1080 outperformed the training speed and next was Xavier and finally Nano. Due to the small number of images and relatively small number of iterations in training, we did all our final training on Nano as the end-to-end turnaround time per test was many times faster.

Training Results

Our first model trained model was providing a 96% accuracy on classifying Free spaces and Blocked spaces, which was better than we expected. The results of the first model are below.

Deploy and Test

The best model from training was saved and loaded onto the robot and when we ran the code. The bot worked sometimes, but also became confused in many cases. A bot's "confusion" looks like taking a couple of good steps before beginning to spin.

Data quality was the main culprit for the bot's confusion. Although the model was at 96% accuracy it did not generalize well. Upon investigation we noticed the quality of data was not high enough. There was very little difference between the images in the 2 classes.

We versioned this model as version 1

Here is a link to the actual video:

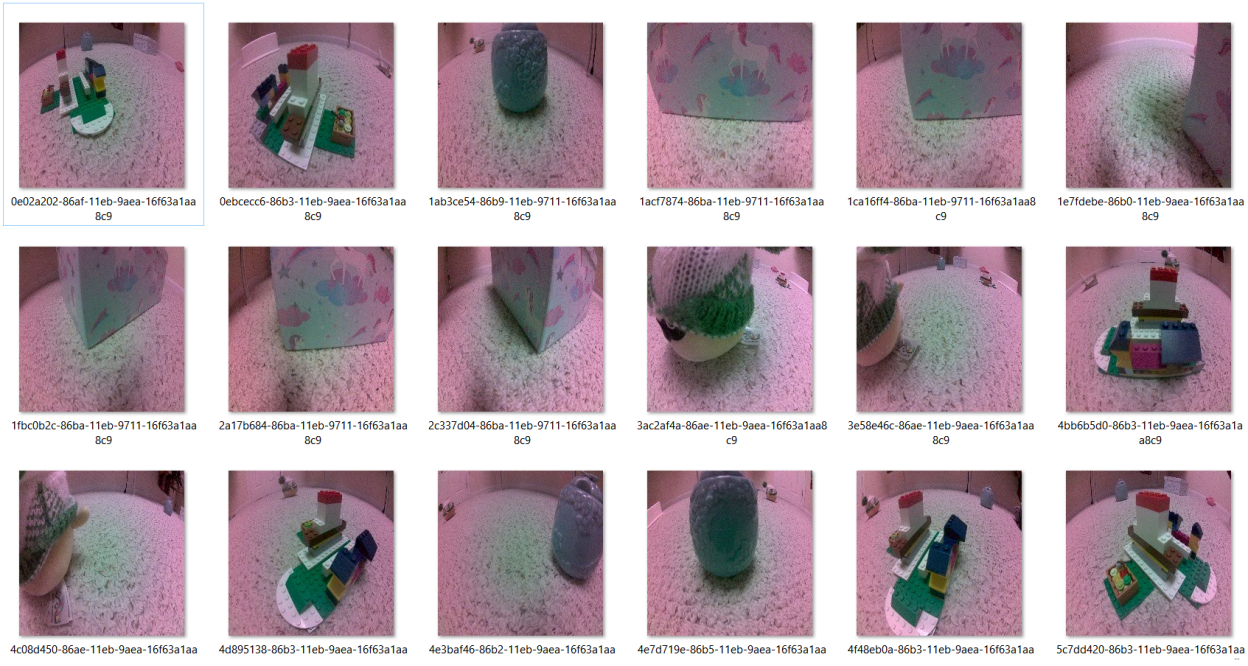
https://drive.google.com/file/d/15oFURD2gBOSZkRTLPkPCnjbnKG0yOT92/view?usp=sharing

Correcting the data

The unexpected behaviour and the diagnosis prompted us to redo the whole data collection process. We changed the environment by moving to a smaller space, removing the green sheets, and most importantly replacing the tiny trinkets with larger objects, and made sure they were labelled correctly

The new Blocked Dataset

We collected 135 images for this class, with a few example images below:

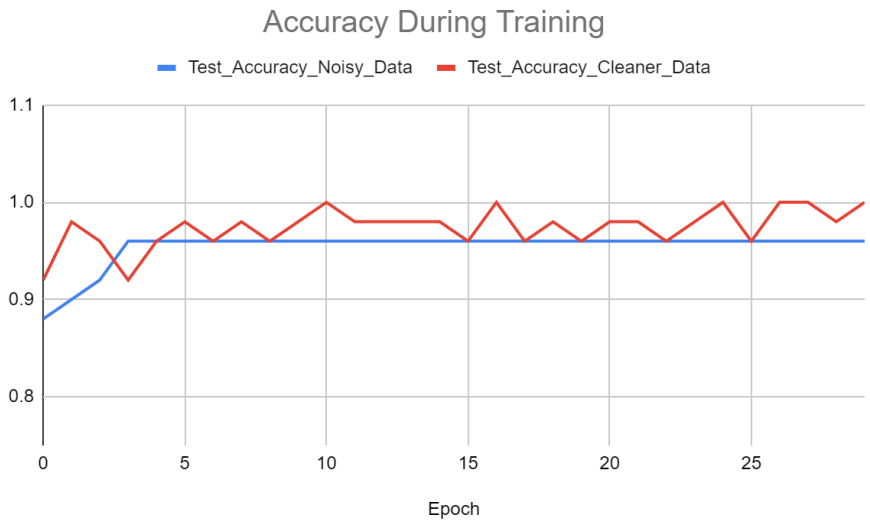


The new Free Dataset

We collected 201 images for this class. Some examples are below.

Training with the new data

We trained the model with the new datasets and all the same hyperparameters. The training accuracy was better than the noisy first version of the data. Below is the chart comparing them both.



Testing the bot with the model v2

The results were significantly better than the model v1. The bot would avoid the obstacles more accurately and drive through the safe spaces . If we let the bot continue , however, we would see the avoiding efficiency decrease, and after 30 seconds it would fail to accurately avoid the obstacles. After a lot of research we realized that the processing of the images was slower than the number of images that were coming in. We used the NVIDIA GStreamer to capture the images and we had to flush the buffer after the bot identified an obstacle. We then had a working bot!

Here is the link to the model v2 video:

https://drive.google.com/file/d/1s_gpyz4hi-JDM113dTMMdK2cOv9Qspcd/view?usp=sharing

Combining Computer Vision and Navigable Space Detection

The final solution had inputs from Navigable Space Detections which fed a list of tuples containing the direction and distance the bot had to travel. We pop the list and follow the direction and distance instructions. While performing the current navigation if an obstacle is detected in the path the rover will go around it.

Here is a link to the final video:

https://drive.google.com/file/d/1MsJBD9QLeUack9GDmJLMIHcfrq5ClYja/view?usp=sharing

Where are we now?

So far, we've got a working, but disjointed pathway from start to finish. We can use an overhead image, find the navigable space, allow the user to change it and choose the "home" space for their rover, and from there, the rover can navigate based on computer vision and the json of navigable space.

What's next?

We need additional reinforcement learning, most likely through simulated machine learning environments. We also need to solve the distance and positioning problem, whether through lidar, gyroscopes and positional sensors, or through calculations based on the property lines and wheel rotations. Additionally, we need to set up most of the data pipeline. Our data needs to go from the overhead image of the address (which we'll likely need a subscription to collect) to the navigable space algorithm, to the user input, to our server, to the mower, and then back to the server to enhance our algorithms. The user input experience also needs to be drastically improved, as the current version is tedious. Lastly, our Rover needs to actually mow! We hope you'll stick around to see the future iterations!